Relatório Trabalho 1 de Lab de Programação Paralela

Alunos: **Kayalla Pontes, Luiz Fellippe Loureiro, Niasi Magalhães**
Repositório no github: **lab-paralela-primeiro-trab**

O chatGPT forneceu o seguinte código:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void gaussElimination(double *matrix, double *b, int n, int rank, int
size) {
    for (int k = 0; k < n; ++k) {
        if (rank == k % size) {
            // Normalizar linha pivô
            double pivot = matrix[k * n + k];
            for (int j = k; j < n; ++j) {
                matrix[k * n + j] /= pivot;
            }
            b[k] /= pivot;
        }

        // Broadcast da linha pivô
        MPI_Bcast(&matrix[k * n], n, MPI_DOUBLE, k % size,
MPI_COMM_WORLD);
        MPI_Bcast(&b[k], 1, MPI_DOUBLE, k % size, MPI_COMM_WORLD);

        // Eliminação das linhas subsequentes
        for (int i = k + 1; i < n; ++i) {
            if (i % size == rank) {
                double factor = matrix[i * n + k];
                for (int j = k; j < n; ++j) {
                    matrix[i * n + j] -= factor * matrix[k * n + j];
                }
                b[i] -= factor * b[k];
            }
        }
    }
}

void backSubstitution(double *matrix, double *b, double *x, int n, int
rank, int size) {
    for (int i = n - 1; i >= 0; --i) {
        if (rank == i % size) {
            x[i] = b[i];
            for (int j = i + 1; j < n; ++j) {
                x[i] -= matrix[i * n + j] * x[j];
            }
        }
```

```
            MPI_Bcast(&x[i], 1, MPI_DOUBLE, i % size, MPI_COMM_WORLD);
        }
    }

    int main(int argc, char *argv[]) {
        int n = 3; // Dimensão do sistema (modifique conforme necessário)
        double matrix[] = {
            2.0, 1.0, -1.0,
            -3.0, -1.0, 2.0,
            -2.0, 1.0, 2.0
        };
        double b[] = {8.0, -11.0, -3.0};
        double x[3];

        int rank, size;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        gaussElimination(matrix, b, n, rank, size);
        backSubstitution(matrix, b, x, n, rank, size);

        if (rank == 0) {
            printf("Solução do sistema:\n");
            for (int i = 0; i < n; ++i) {
                printf("x[%d] = %f\n", i, x[i]);
            }
        }

        MPI_Finalize();
        return 0;
    }
```

A primeira coisa que fizemos foi adicionar a função MPI_Wtime() para marcar o tempo de execução.

```
tempo_inicial = MPI_Wtime();
gaussElimination(matrix, b, n, rank, size);
backSubstitution(matrix, b, x, n, rank, size);
tempo_final = MPI_Wtime();
```

Os tempos iniciais foram muito pequenos, porém os tempos para mais processos estava dando maior. Isso se dá pelo overhead de comunicação, a própria divisão de processos acabou usando mais tempo:

```
$ mpirun -np 2 ./gauss_mpi_melhorado
Solução do sistema:
x[0] = 2.000000
x[1] = 3.000000
```

```
x[2] = -1.000000
Foram gastos 0.0000430000 segundos

$ mpirun -np 4 ./gauss_mpi_melhorado
Solução do sistema:
x[0] = 2.000000
x[1] = 3.000000
x[2] = -1.000000
Foram gastos 0.0000680000 segundos

$ mpirun -np 8 ./gauss_mpi_melhorado
Solução do sistema:
x[0] = 2.000000
x[1] = 3.000000
x[2] = -1.000000
Foram gastos 0.0001970000 segundos
```

Então, ao invés de criar uma pequena matriz estática, criamos dinamicamente uma matriz de 2000 valores (para os parâmetros do computador em que foi rodado, é um valor médio). Para isso, usamos malloc para armazenar essa matriz.

```
    int n = 2000; // Tamanho do sistema
    double *matrix, *b, *x;

    // Alocação dinâmica dos endereços para a matriz
    matrix = (double *)malloc(n * n * sizeof(double));
    b = (double *)malloc(n * sizeof(double));
    x = (double *)malloc(n * sizeof(double));

    if (!matrix || !b || !x) {
        printf("Erro na alocação de memória\n");
        exit(1);
    }

    // Preenche matriz e vetor com valores aleatórios
    srand(time(NULL));
    for (int i = 0; i < n; ++i) {
        b[i] = rand() % 100; // Valores no vetor b
        for (int j = 0; j < n; ++j) {
            matrix[i * n + j] = (i == j) ? rand() % 100 + 1 : rand() %
100;
            // Matriz diagonal dominante
        }
    }
```

Para conferir se o resultado encontrado para a matriz x está correto, comparamos ax (matriz inicial * resultado encontrado) com o b aleatório inicial:

```
// Verificando a precisão da solução
double error = 0.0;
for (int i = 0; i < n; ++i) {
    double ax = 0.0;
    for (int j = 0; j < n; ++j) {
        ax += matrix[i * n + j] * x[j]; // calcula os resultados para b
com x encontrado
    }
    error += fabs(ax - b[i]); // compara o valor com o b aleatório
inicial
}
printf("Erro total: %e\n", error);
```

Obtivemos os seguintes resultados:

```
$ mpirun -np 2 ./gauss_mpi_melhorado
Foram gastos 2.0763030000 segundos
Erro total: 2.814443e-10

$ mpirun -np 4 ./gauss_mpi_melhorado
Foram gastos 1.1009420000 segundos
Erro total: 1.766998e-10

$ mpirun -np 8 ./gauss_mpi_melhorado
Foram gastos 2.2496170000 segundos
Erro total: 6.402490e-10
```

Notamos que a função `backSubstitution` tinha um erro: no final, o valor de x[i] não estava sendo normalizado pelo coeficiente da diagonal principal, o que pode levar a resultados incorretos. Adicionamos então a linha `x[i] /= matrix[i * n + i];` para consertar. Isso diminuiu um pouco o erro:

```
$ mpirun -np 2 ./gauss_mpi_melhorado
Foram gastos 2.0777840000 segundos
Erro total: 1.280451e-10

$ mpirun -np 4 ./gauss_mpi_melhorado
Foram gastos 1.1075860000 segundos
Erro total: 1.769355e-10

$ mpirun -np 8 ./gauss_mpi_melhorado
Foram gastos 2.1546820000 segundos
Erro total: 3.593319e-10
```

Depois disso, tentamos rodar para um valor um pouco mais extremo, uma matriz de n = 5.000. O tempo aumentou consideravelmente.

Como o computador era quadcore, tomamos a liberdade de ao invés de usar 2, 4 e 8, usar 1, 2 e 4, pois nos entregava resultados mais satisfatórios em relação ao tempo.

Resultado para n = 100:

```
$ mpirun -np 1 ./gauss_mpi_melhorado
Foram gastos 0.0012420000 segundos
Erro total: 3.888513e-13

$ mpirun -np 2 ./gauss_mpi_melhorado
Foram gastos 0.0017670000 segundos
Erro total: 1.271587e-13

$ mpirun -np 4 ./gauss_mpi_melhorado
Foram gastos 0.0017940000 segundos
Erro total: 5.108414e-13
```

Resultado para n = 2000:

```
$ mpirun -np 1 ./gauss_mpi_melhorado
Foram gastos 3.9990370000 segundos
Erro total: 2.640805e-10

$ mpirun -np 2 ./gauss_mpi_melhorado
Foram gastos 2.0781060000 segundos
Erro total: 2.371149e-10

$ mpirun -np 4 ./gauss_mpi_melhorado
Foram gastos 1.1092260000 segundos
Erro total: 9.125933e-11
```

Resultado para n = 5000:

```
$ mpirun -np 1 ./gauss_mpi_melhorado
Foram gastos 62.5402540000 segundos
Erro total: 6.083387e-10

$ mpirun -np 2 ./gauss_mpi_melhorado
Foram gastos 32.6793270000 segundos
Erro total: 4.035896e-09

$ mpirun -np 4 ./gauss_mpi_melhorado
Foram gastos 17.2685780000 segundos
Erro total: 3.014510e-09
```

Substituímos o MPI_Bcast pelo uso combinado de MPI_Reduce seguido de MPI_Bcast no algoritmo de eliminação de Gauss para otimizar a comunicação entre os processos. Após a redução, o MPI_Bcast é utilizado para disseminar os resultados da soma apenas uma vez, garantindo que todos os processos tenham acesso aos valores necessários.

```
void gaussElimination(double *matrix, double *b, int n, int rank, int
size) {
    for (int k = 0; k < n; ++k) {
        if (rank == k % size) {
            // Normalizar linha pivô
            double pivot = matrix[k * n + k];
            for (int j = k; j < n; ++j) {
                matrix[k * n + j] /= pivot;
            }
            b[k] /= pivot;
        }

        // Reduzir linha pivô para todos os processos
        double *pivot_row = (double *)malloc(n * sizeof(double));
        MPI_Reduce(&matrix[k * n], pivot_row, n, MPI_DOUBLE, MPI_SUM, k
% size, MPI_COMM_WORLD);

        double pivot_b;
        MPI_Reduce(&b[k], &pivot_b, 1, MPI_DOUBLE, MPI_SUM, k % size,
MPI_COMM_WORLD);

        // Broadcast dos valores reduzidos (somente o processo
responsável envia)
        MPI_Bcast(pivot_row, n, MPI_DOUBLE, k % size, MPI_COMM_WORLD);
        MPI_Bcast(&pivot_b, 1, MPI_DOUBLE, k % size, MPI_COMM_WORLD);

        // Substituir valores locais pela linha pivô recebida
        for (int j = 0; j < n; ++j) {
            matrix[k * n + j] = pivot_row[j];
        }
        b[k] = pivot_b;

        free(pivot_row);

        // Eliminação das linhas subsequentes
        for (int i = k + 1; i < n; ++i) {
            if (i % size == rank) {
                double factor = matrix[i * n + k];
                for (int j = k; j < n; ++j) {
                    matrix[i * n + j] -= factor * matrix[k * n + j];
                }
                b[i] -= factor * b[k];
            }
        }
    }
}
```

Fizemos o teste utilizando n = 2000, caso médio. Porém, isso não mudou tanto o tempo de execução de cada um.

```
$ mpirun -np 1 ./gauss_mpi_melhorado
Foram gastos 3.9976750000 segundos

$ mpirun -np 2 ./gauss_mpi_melhorado
Foram gastos 2.0999380000 segundos

$ mpirun -np 4 ./gauss_mpi_melhorado
Foram gastos 1.1897240000 segundos
```

Substituímos então tudo por por MPI_Allreduce, que elimina a necessidade de um MPI_Reduce seguido por MPI_Bcast, unindo as duas operações em uma única chamada de função. O MPI_Allreduce realiza a redução dos dados em todos os processos e distribui simultaneamente o resultado para todos.

```
void gaussElimination(double *matrix, double *b, int n, int rank, int
size) {
    double *pivot_row = (double *)malloc(n * sizeof(double));

    for (int k = 0; k < n; ++k) {
        // Reduzir e compartilhar simultaneamente a linha pivô com
MPI_Allreduce
        MPI_Allreduce(&matrix[k * n], pivot_row, n, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

        // Reduzir e compartilhar o elemento de b correspondente
        double pivot_b;
        MPI_Allreduce(&b[k], &pivot_b, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

        if (rank == k % size) {
            // Atualiza a linha pivô no processo raiz para manter
consistência local
            for (int j = 0; j < n; ++j) {
                matrix[k * n + j] = pivot_row[j];
            }
            b[k] = pivot_b;
        }

        // Eliminação das linhas subsequentes
        for (int i = k + 1; i < n; ++i) {
            if (i % size == rank) {
                double factor = matrix[i * n + k];
                for (int j = k; j < n; ++j) {
                    matrix[i * n + j] -= factor * pivot_row[j];
```

```
                }
                b[i] -= factor * pivot_b;
            }
        }
    }

    free(pivot_row);
}
```

Os resultados de tempo foram um pouco melhores agora:

```
$ mpirun -np 1 ./gauss_mpi_melhorado
Foram gastos 3.4877870000 segundos

$ mpirun -np 2 ./gauss_mpi_melhorado
Foram gastos 1.8409990000 segundos

$ mpirun -np 4 ./gauss_mpi_melhorado
Foram gastos 0.9792150000 segundos
```

Testamos então com o MPI_Scatter para distribuir a matriz para os processos, de forma que cada processo tenha uma parte da matriz para trabalhar.
Não há necessidade de um MPI_Gather explícito, já que as atualizações nos vetores x podem ser compartilhadas com MPI_Bcast.

```
void gaussElimination(double *matrix, double *b, int n, int rank, int
size) {
    double *pivot_row = (double *)malloc(n * sizeof(double));

    for (int k = 0; k < n; ++k) {
        // Distribui a linha pivô para todos os processos com
MPI_Scatter
        MPI_Scatter(&matrix[k * n], n / size, MPI_DOUBLE, pivot_row, n /
size, MPI_DOUBLE, k % size, MPI_COMM_WORLD);

        // Reduzir e compartilhar o elemento de b correspondente
        double pivot_b;
        MPI_Bcast(&b[k], 1, MPI_DOUBLE, k % size, MPI_COMM_WORLD);

        if (rank == k % size) {
            // Atualiza a linha pivô no processo raiz para manter
consistência local
            for (int j = 0; j < n; ++j) {
                matrix[k * n + j] = pivot_row[j];
            }
            b[k] = pivot_b;
        }
```

```
        // Eliminação das linhas subsequentes
        for (int i = k + 1; i < n; ++i) {
            if (i % size == rank) {
                double factor = matrix[i * n + k];
                for (int j = k; j < n; ++j) {
                    matrix[i * n + j] -= factor * pivot_row[j];
                }
                b[i] -= factor * pivot_b;
            }
        }
    }

    free(pivot_row);
}
```

Este código deu o mesmo tempo que o anterior.

```
$ mpirun -np 1 ./gauss_mpi_melhorado
Foram gastos 3.4971430000 segundos

$ mpirun -np 2 ./gauss_mpi_melhorado
Foram gastos 1.8210440000 segundos

$ mpirun -np 4 ./gauss_mpi_melhorado
Foram gastos 0.9705120000 segundos
```

A análise de speedup e eficiência está em Análises Speedup.